Aufgabe 1

Die Syntax von WHILE sei um die Regel

```
C::= repeat C until B
```

erweitert. Ergänzen Sie die operationelle Semantik von WHILE, so dass diese zusätzliche Anweisungsform angemessen behandelt wird.

$\Delta\langle W|S|\underline{repeat}\ C\ \underline{until}\ B.K|E|A\rangle = \langle W|S|C.\underline{while}\ (\underline{not} B)\ \underline{do}\ C.K|E|A\rangle$
C wird mindestens einmal ausgeführt, anschliessend verhält sich die „repeat C until B"-Schleife identisch zu „while ¬B do C ".

Aufgabe 2

Erweitern Sie die Syntax von WHILE, so dass in den boolschen Ausdrücken auch die boolschen Operatoren *and* und *or* vorkommen dürfen. Geben Sie für diese Erweiterung eine operationelle Semantik an, die eine nicht-strikte Semantik von *and* und *or* festlegt.

$B ::= W|\underline{not}\ B|T_1\ BOP\ T_2|B_1\ \underline{and}\ B_2|B_1\ \underline{or}\ B_2|\underline{read}$

And:
$\Delta\langle W|S|B_1\ \underline{and}\ B_2.K|E|A\rangle = \langle W|S|B_1.\underline{and}.B_2.K|E|A\rangle$
$\Delta\langle \underline{true}.W|S|\underline{and}.B_2.K|E|A\rangle = \langle W|S|B_2.K|E|A\rangle$
$\Delta\langle \underline{false}.W|S|\underline{and}.B_2.K|E|A\rangle = \langle \underline{false}.W|S|K|E|A\rangle$

Or:
$\Delta\langle W|S|B_1\ \underline{or}\ B_2.K|E|A\rangle = \langle W|S|B_1.\underline{or}.B_2.K|E|A\rangle$
$\Delta\langle \underline{false}.W|S|\underline{or}.B_2.K|E|A\rangle = \langle W|S|B_2.K|E|A\rangle$
$\Delta\langle \underline{true}.W|S|\underline{or}.B_2.K|E|A\rangle = \langle \underline{true}.W|S|K|E|A\rangle$

Aufgabe 3

Erweitern Sie die WSKEA-Maschine um eine Komponente N für Nachrichten (Texte), in der kurze, sinnvolle Meldungen eingetragen werden, wenn es keinen Folgezustand gibt, oder wenn die Ausführung korrekt terminiert.

Grundsätzlich identisch zur WSKEA-Maschine

Grundzustand der WSKEAN-Maschine ist: $\langle W|S|K|E|A|\epsilon\rangle$

Falls das Programm korrekt terminiert gilt: $\Delta\langle W|S|\epsilon|E|A|\epsilon\rangle = \langle W|S|\epsilon|E|A|$„Terminiert"$\rangle$

In einem Fehlerzustand gibt die Maschine eine informative Fehlermeldung zurück. Ein Beispiel wäre:

$\Delta\langle 0.n.W|S|/.K|E|A|\epsilon\rangle = \langle W|S|K|E|A|$„Fehler: Division durch o"$\rangle$

Die Darstellung weiterer Fehlerzustände ist analog dazu.

Aufgabe 4 (freiwillig)

Implementieren Sie in einer Sprache Ihrer Wahl

- den Zustandsraum der WSKEA-Maschine,
- eine Funktion `anfang`, die zu einem WHILE-Programm und einer Eingabe den Anfangszustand ergibt, und
- die Zustandsüberführungsfunktion `delta`.

Lösung in **Whitespace** ;-) :

Lösungsidee in **Python**:

```
__author__ = 'TH'

# anfang
w = ['x']
s = []
k = ['output', '=', 1, 1, 'not', True, 'add', 'read', 'read', 'skip', '
    assign', 42]
e = [1, 1]
a = []
test = [w, s, k, e, a]


def run(machine):   # delta
    w = machine[0]  # process stack
    s = machine[1]  # variable store
    k = machine[2]  # cmd stack
    e = machine[3]  # input (file)
    a = machine[4]  # output (file)
    i = 0
    while len(k) > 0:
        cmd = k.pop()
        if type(cmd) == int:
            w.append(cmd)
        elif type(cmd) == bool:
            w.append(cmd)
        elif cmd == 'not':
            b = not w.pop()
            w.append(b)
        elif cmd == '=':
            w.append(w.pop() == w.pop())
        elif cmd == 'read':
            w.append(e.pop())
        elif cmd == 'add' or cmd == '+':
            w.append(w.pop() + w.pop())
        elif cmd == 'sub' or cmd == '-':
            w.append(w.pop() - w.pop())
        elif cmd == 'mul' or cmd == '*':
            w.append(w.pop() * w.pop())
        elif cmd == 'div'or cmd == '/':
            w.append(w.pop() / w.pop())
        elif cmd == 'skip':
            pass
        elif cmd == 'assign':
            s.append((w.pop(), w.pop()))
        elif cmd == 'output':
            a.append(w.pop())
        print('run ' + str(i) + ': ' + str(machine))
        i += 1
    print('success after '+str(i)+' iterations !')
    return 1


# LETS DO THIS!

def main():
    return run(test)

main()
```

Ausgabe:

```
run 0: [['x', 42], [], ['output', '=', 1, 1, 'not', True, 'add', 'read', '
    read', 'skip', 'assign'], [1, 1], []]
run 1: [[], [(42, 'x')], ['output', '=', 1, 1, 'not', True, 'add', 'read',
    'read', 'skip'], [1, 1], []]
run 2: [[], [(42, 'x')], ['output', '=', 1, 1, 'not', True, 'add', 'read',
    'read'], [1, 1], []]
run 3: [[1], [(42, 'x')], ['output', '=', 1, 1, 'not', True, 'add', 'read'
    ], [1], []]
run 4: [[1, 1], [(42, 'x')], ['output', '=', 1, 1, 'not', True, 'add'], [],
     []]
run 5: [[2], [(42, 'x')], ['output', '=', 1, 1, 'not', True], [], []]
run 6: [[2, True], [(42, 'x')], ['output', '=', 1, 1, 'not'], [], []]
run 7: [[2, False], [(42, 'x')], ['output', '=', 1, 1], [], []]
run 8: [[2, False, 1], [(42, 'x')], ['output', '=', 1], [], []]
run 9: [[2, False, 1, 1], [(42, 'x')], ['output', '='], [], []]
run 10: [[2, False, True], [(42, 'x')], ['output'], [], []]
run 11: [[2, False], [(42, 'x')], [], [], [True]]
success after 12 iterations !
```

Kommentar zur Lösung:
Die Lösung nutzt einige Annehmlichkeiten von Python: Listen und Typen. Trotzdem sind nicht alle Konstrukte implementiert. Bisher habe ich noch keine Idee, wie ich $I, C, T, \underline{BOP}, \underline{OP}$ in Python repräsentiere, daher sind einige Ausdrücke in diesem Programm nicht ausführbar. Damit wird es sehr schwierig Strukturen wie `if B then C1 else C2` oder `while B do C` zu implementieren und damit die Aufgabe nicht vollständig bearbeitet.